**Приложение В**

**Грамматика языка Proto**

**Грамматика языка для ANTLR**

grammar Proto;

options { output=AST; }

@header {

package proto.antlr;

import proto.antlr.ClassNode;

import proto.antlr.MethodNode;

import proto.antlr.StateNode;

import proto.antlr.CallNode;

import proto.antlr.PrototypeNode;

import proto.antlr.SpecNode;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Stack;

}

@lexer::header {

package proto.antlr;

}

@members {

Stack<HashMap<String, String>> scope = new Stack<HashMap<String, String>>() {{}};

String scp;

}

program

: ( statement )\*

;

statement

: interface\_decl

| class\_decl

| proto\_decl

;

proto\_decl

: 'prototype' ID '(' parameters ')' '{' ( spec\_decl )\* '}'

-> ^(ID<PrototypeNode> ( spec\_decl )\* )

;

spec\_decl

: UNARY\_SPEC+ '{' spec\_expression '}' ';'

-> ^(UNARY\_SPEC<SpecNode> UNARY\_SPEC\* '{' spec\_expression '}' )

| '{' spec\_expression '}' BINARY\_SPEC '{' spec\_expression '}' ';'

-> ^('{'<SpecNode> spec\_expression '}' BINARY\_SPEC '{' spec\_expression '}')

;

UNARY\_SPEC

: 'finally'

| 'globally'

;

BINARY\_SPEC

: 'until'

| 'release'

| 'if'

;

spec\_expression

: or\_spec

-> or\_spec

;

or\_spec

: and\_spec ( 'or' and\_spec )?

;

and\_spec

: not\_spec ( 'and' not\_spec )?

;

not\_spec

: ( 'not' )? spec

;

spec

: one\_spec ( '==' | '!=' one\_spec )\*

;

one\_spec

: ID -> ID<SpecNode>

| '(' spec\_expression ')' -> '(' spec\_expression? ')'

;

interface\_decl

: 'interface' ID '{' ( method\_decl )\* '}'

->

;

method\_decl

: type ID '(' arguments ')' ';'

;

arguments

: ( t=type i=ID

{

if(!scope.isEmpty())

scope.peek().put($i.text, $t.text);

}

( ',' t=type i=ID

{

scope.peek().put($i.text, $t.text);

}

)\* )?

;

parameters

: ( big\_expression ( ',' big\_expression )\* )?

-> big\_expression? (big\_expression)?

;

type

: 'void'

| 'state' array

| 'number' array

| 'bool' array

| 'string' array

| 'object' array

| ID array

;

array

: ( '[' ']' )\*

;

class\_decl

: 'class' i=ID '(' arguments ')' ( '<' ID ( ',' ID )\* )? ( '<<' ID )? '{'

{

scope.push(new HashMap<String, String>());

scope.peek().put("self", $i.text);

}

(body)\*

{

scope.pop();

}

'}'

-> ^(ID<ClassNode> (body)\*)

;

body

: method

| field

;

method

: t=type ID

{

String classname = scope.peek().get("self");

scope.push(new HashMap<String, String>());

scope.peek().put("self", classname);

}

'(' arguments ')' '{' ( operator )\* '}'

{

scope.pop();

}

-> {$t.text.equals("state")}?

^(ID<StateNode>[$ID, scope.peek().get("self")] (operator)\* )

-> ^(ID<MethodNode>[$ID, scope.peek().get("self")] (operator)\*)

;

field

: t=type i=ID

{

scope.peek().put($i.text, $t.text);

}

( '=' big\_expression )? ';'

-> big\_expression?

;

operator

: field

| assignment ';' -> assignment

| buildin\_operator

| call ';'

{ scp=null; } -> call

| if\_operator

| for\_operator

| while\_operator

| do\_operator

| '{' ( operator )\* '}' -> (operator)\*

;

buildin\_operator

: print

| die

| return\_operator

;

die

: 'die' big\_expression ';'

-> big\_expression?

;

print

: 'print' big\_expression ';'

-> big\_expression?

;

return\_operator

: 'return' big\_expression ';'

-> big\_expression?

;

assignment

: ID ( '[' big\_expression ']' )\* '=' big\_expression

-> (big\_expression)\*

;

big\_expression

: or\_expression

;

or\_expression

: and\_expression ( 'or' and\_expression )?

-> (and\_expression)\*

;

and\_expression

: not\_expression ( 'and' not\_expression )?

-> (not\_expression)\*

;

not\_expression

: ( 'not' )? expression

-> expression?

;

expression

: relation ( ( '==' | '!=' ) relation )?

-> (relation)\*

;

relation

: summand ( ('>' | '<' | '<=' | '>=') summand )?

-> (summand)\*

;

summand

: multiplier ( '+' multiplier | '-' multiplier )\*

-> (multiplier)\*

;

multiplier

: simple\_expression ( '\*' simple\_expression | '/' simple\_expression )\*

-> (simple\_expression)\*

;

simple\_expression

: ID ( '[' big\_expression ']' )\* ->

| call

| INT ->

| STRING ->

| '[' parameters ']' -> parameters?

| '(' big\_expression ')' -> big\_expression?

| 'nan' ->

| 'nil' ->

| 'new' ID '(' parameters ')' -> parameters?

| 'random' ( ID | INT ) ->

;

call

: i=ID '(' parameters ')'

{

if($i.text.indexOf(".") != -1) { // if . in call

for(int j=scope.size()-1; j>=0; j--) {

if(!scope.get(j).isEmpty()) {

String id = scope.get(j).get($i.text.substring(0, $i.text.indexOf(".")));

if(id != null) {

scp=id; break;

}

}

}

} else {

scp = scope.peek().get("self");

}

}

-> ^( ID<CallNode>[$ID, scp] parameters? )

;

if\_operator

: 'if' '(' big\_expression ')' operator ( 'else' operator )?

-> big\_expression? (operator)\*

;

for\_operator

: 'for' '(' assignment ';' big\_expression ';' assignment ')' operator

-> assignment big\_expression? assignment operator

;

while\_operator

: 'while' '(' big\_expression ')' operator

-> big\_expression? operator

;

do\_operator

: 'do' '{' operator '}' 'while' '(' big\_expression ')' ';'

-> operator big\_expression?

;

ID : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_'|'::'|'.')\*

;

INT : '0'..'9'+

;

COMMENT

: '//' ~('\n'|'\r')\* '\r'? '\n' {$channel=HIDDEN;}

| '/\*' ( options {greedy=false;} : . )\* '\*/' {$channel=HIDDEN;}

;

WS : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

STRING

: '"' ( ESC\_SEQ | ~('\\'|'"') )\* '"'

;

fragment

HEX\_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment

ESC\_SEQ

: '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')

| UNICODE\_ESC

| OCTAL\_ESC

;

fragment

OCTAL\_ESC

: '\\' ('0'..'3') ('0'..'7') ('0'..'7')

| '\\' ('0'..'7') ('0'..'7')

| '\\' ('0'..'7')

;

fragment

UNICODE\_ESC

: '\\' 'u' HEX\_DIGIT HEX\_DIGIT HEX\_DIGIT HEX\_DIGIT

;

**Грамматика языка для XTEXT**

grammar org.example.domainmodel.DomainModel with org.eclipse.xtext.common.Terminals

generate domainModel "http://www.example.org/domainmodel/DomainModel"

Domainmodel:

elements+=Statement\*;

Statement:

Interface\_decl | Class\_decl | Proto\_decl;

QualifiedName:

ID (('\_'|'::'|'.') ID)\*;

Proto\_decl:

'prototype' name=QualifiedName '(' params=Parameters ')' '{' specs+=Spec\_decl\* '}';

Spec\_decl:

UNARY\_SPEC+ '{' Or\_spec '}' ';'

| '{' Or\_spec '}' BINARY\_SPEC '{' or=Or\_spec '}' ';';

UNARY\_SPEC:

'finally'

| 'globally';

BINARY\_SPEC:

'until'

| 'release'

| 'if';

Or\_spec:

And\_spec ('or' ands=And\_spec)?;

And\_spec:

Not\_spec ('and' nots=Not\_spec)?;

Not\_spec:

('not')? Spec;

Spec:

One\_spec ('==' | '!=' ones+=One\_spec)\*;

One\_spec:

state=QualifiedName | '(' spece=Or\_spec ')';

Class\_decl:

'class' name=QualifiedName '(' classargs=Arguments ')' ('<' implements1=QualifiedName (','

implements2+=QualifiedName)\*)? ('<<' superType=QualifiedName)? '{' (classbody+=Body)\* '}';

Interface\_decl:

'interface' iname=QualifiedName '{' (intfbode+=Method\_decl)\* '}';

Method\_decl:

type=Types name=QualifiedName '(' args=Arguments ')' ';';

Arguments:

(type=Types argname=QualifiedName (',' type=Types argname=QualifiedName)\*)?;

Parameters:

(params+=Big\_expression (',' params+=Big\_expression)\*)?;

Types:

'void'

| 'state' Array

| 'number' Array

| 'bool' Array

| 'string' Array

| 'object' Array

| QualifiedName Array;

Array:

('[' ']')\*;

Operator:

Field

| Assignment ';'

| Buildin\_operator

| Call ';'

| If\_operator

| For\_operator

| While\_operator

| Do\_operator

| '{' newblock+=Operator\* '}';

Buildin\_operator:

Print

| Die

| Return\_operator;

Die:

'die' Big\_expression ';';

Print:

'print' Big\_expression ';';

Return\_operator:

'return' Big\_expression ';';

Assignment:

name=QualifiedName ('[' index+=Big\_expression ']')\* '=' value=Big\_expression;

Big\_expression:

And\_expression ('or' ande=And\_expression)?;

And\_expression:

Not\_expression ('and' note=Not\_expression)?;

Not\_expression:

('not')? Expression;

Expression:

Relation (('==' | '!=') rel=Relation)?;

Relation:

Summand (('>' | '<' | '<=' | '>=') sum=Summand)?;

Summand:

Multiplier (('+' | '-') mul=Multiplier)\*;

Multiplier:

Simple\_expression (('\*' | '/') se=Simple\_expression)\*;

Field:

type=Types name=QualifiedName ('=' value=Big\_expression)? ';';

Simple\_expression:

name = QualifiedName ('[' index+=Big\_expression ']')\*

| Call

| INT

| STRING

| '[' params=Parameters ']'

| '(' be=Big\_expression ')'

| 'nan'

| 'nil'

| 'new' name=QualifiedName '(' params=Parameters ')'

| 'random' (QualifiedName | INT);

Body:

Method

| Field;

Method:

type=Types name=QualifiedName '(' args=Arguments ')' '{' (methodbody+=Operator)\* '}';

Call:

name=QualifiedName '(' params=Parameters ')';

If\_operator:

'if' '(' ifcond=Big\_expression ')' ifbody=Operator ('else' elsebody=Operator)?;

For\_operator:

'for' '(' Assignment ';' forcond=Big\_expression ';' forassign=Assignment ')' bode=Operator;

While\_operator:

'while' '(' Big\_expression ')' whilebody=Operator;

Do\_operator:

'do' '{' dobody=Operator '}' 'while' '(' docond=Big\_expression ')' ';';